Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters

Clémentine Maurice[1,2] (✉), Nicolas Le Scouarnec[1], Christoph Neumann[1], Olivier Heen[1], Aurélien Francillon[2]

[1] Technicolor, Rennes, France
[2] Eurecom, Sophia Antipolis, France

Abstract. Cache attacks, which exploit differences in timing to perform covert or side channels, are now well understood. Recent works leverage the last level cache to perform cache attacks across cores. This cache is split in *slices*, with one slice per core. While predicting the slices used by an address is simple in older processors, recent processors are using an undocumented technique called *complex addressing*. This renders some attacks more difficult and makes other attacks impossible, because of the loss of precision in the prediction of cache collisions.
In this paper, we build an automatic and generic method for reverse engineering Intel's last-level cache complex addressing, consequently rendering the class of cache attacks highly practical. Our method relies on CPU hardware performance counters to determine the cache slice an address is mapped to. We show that our method gives a more precise description of the complex addressing function than previous work. We validated our method by reversing the complex addressing functions on a diverse set of Intel processors. This set encompasses Sandy Bridge, Ivy Bridge and Haswell micro-architectures, with different number of cores, for mobile and server ranges of processors. We show the correctness of our function by building a covert channel. Finally, we discuss how other attacks benefit from knowing the complex addressng of a cache, such as sandboxed *rowhammer*.

Keywords: Complex addressing, Covert channel, Cross-Core, Last level cache, Reverse engineering, Side channel.

1 Introduction

In modern x86 micro-architectures, the cache is an element that is shared by cores of the same processor. It is thus a piece of hardware of choice for performing attacks. Cache attacks like covert and side channels can be performed in virtualized environments [16, 22, 27, 33, 35–37], breaching the hypervisor isolation at the hardware level. Caches are also exploited in other types of attacks, such as bypassing kernel ASLR [8], or detecting cryptographic libraries in virtualized environments [17].

Cache attacks are based on difference of timings: the access to a cached memory line is fast, while the access to a previously evicted cache line is slow. Cache attacks can operate at all cache levels: level 1 (L1), level 2 (L2) and Last Level Cache (LLC). Attacks on the L1 or L2 cache restrict the attacker to be on the same core as the victim. This is a too strong assumption on a multi-core processor when the attacker and the victim migrate across cores [27, 35]. We thus focus on cache attacks on the last level cache, which is shared among cores in modern processors. Attacks on the last level cache are more powerful as the attacker and the victim can run on different cores, but they are also more challenging. To perform these attacks, the attacker has to target specific sets in the last level cache. He faces two issues: the last level cache is physically addressed, and modern processors map an address to a slice using the so-called *complex addressing* scheme which is undocumented.

A first set of attacks requires shared memory and evicts a specific line using the `clflush` instruction [36, 16, 38, 7]. However, a simple countermeasure to thwart such side channels is to disable memory sharing across VMs, which is already done by most cloud providers.

Without using any shared memory, an attacker has to find addresses that map to the same set, and exploit the cache replacement policy to evict lines. On processors that do not use complex addressing, huge pages are sufficient to enable side channels by targeting a precise set [14]. On recent processors that use complex addressing, this difficulty can be bypassed by evicting the whole LLC [22], but the temporal resolution makes it impossible to perform side channels. Liu et al. [20] and Oren et al. [24] construct eviction sets by seeking conflicting addresses, enabling fine-grained covert and side channels. This works without reverse engineering the complex addressing function, but has to be performed for each attack.

Hund et al. [8] manually and, as we show, only partially reverse engineered the complex addressing function to defeat kernel ASLR on a Sandy Bridge processor. The challenge in reversing the complex addressing function is to retrieve all the bits. Indeed, previous approaches rely on timing attacks with conflicting cache sets. As the set bits are fixed, they cannot be retrieved this way. Previous work was also incomplete because the function differs for processors with different numbers of cores, as we will show.

Reversing this addressing function also gains momentum [29] in discussions about the exploitation of the so-called *rowhammer* vulnerability. Indeed, *rowhammer* can cause random bit flips in DRAM chips by accessing specific memory locations repeatedly [19]. The exploitation of this vulnerability uses the `clflush` instruction [28]. This instruction has been disabled [3] in the Native Client sandbox [2] due to this security issue. Reversing the addressing function could lead to new ways to exploit *rowhammer* without relying on the `clflush` instruction.

In this paper, we automate reverse engineering of the complex cache addressing in order to make these attacks more practical. In contrast to previous work that reverse engineered the function manually, we develop a fully automatic approach to resolve the complex addressing of last level cache slices. Our technique

relies on performance counters to measure the number of accesses to a slice and to determine on which slice a memory access is cached. As a result, we obtain a translation table that allows determining the slice used by a given physical address (Section 3). In the general case, finding a compact function from the mapping is NP-hard. Nevertheless, we show an efficient algorithm to find a compact solution for a majority of processors (which have 2^n cores). As a result, we provide new insights on the behavior of the last level cache, and refine many previous works (e.g., Hund et al. [8]). In particular, we obtain a more complete and more precise description than previous work, *i.e.*, taking into account more bits of the memory address and fixing the partially incorrect functions of prior work. We evaluate our method on processors of different micro-architectures with various numbers of cores (Section 4). We demonstrate the correctness of our function by building a *prime+probe* covert channel (Section 5). Finally, we discuss the difference between our findings and the previous attempts of reverse engineering this function, as well as other applications (Section 6).

Contributions

In summary, this paper presents the following main contributions:

1. We introduce a generic method for mapping physical addresses to last level cache slices, using hardware performance counters.
2. We provide a compact function for most processor models (with 2^n cores).
3. We validate our approach on a wide range of modern processors.
4. We show, and discuss, practical examples of the benefits to cache attacks.

2 Background

In this section, we give details on cache internals for Intel processors post Sandy Bridge micro-architecture (2011). We then review attacks that exploit cache interferences to perform covert and side channels. Finally, we provide background on hardware performance counters.

2.1 Cache Fundamentals

The processor stores recently-used data in a hierarchy of caches to reduce the memory access time by the processor (see Figure 1). The first two levels L1 and L2 are usually small and private to each core. The L3 is also called Last Level Cache (LLC). It is shared among cores and can store several megabytes. The LLC is *inclusive*, which means it is a superset of the lower levels.

Caches are organized in 64-byte long blocks called *lines*. The caches are *n-way associative*, which means that a line is loaded in a specific set depending on its address, and occupies any of the n lines. When all lines are used in a set, the *replacement policy* decides the line to be evicted to make room for storing a new cache line. Efficient replacement policies favor lines that are the least likely to be reused. Such policies are usually variations of Least Recently Used (LRU).

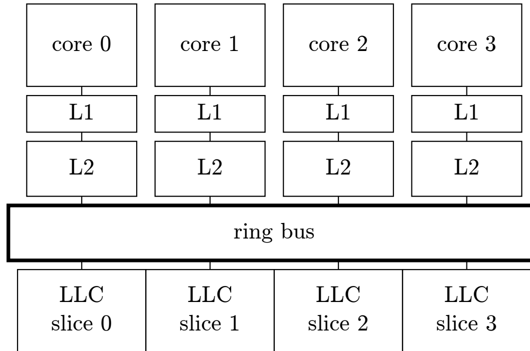

Fig. 1: Cache architecture of a quad-core Intel processor (since Sandy Bridge micro-architecture). The LLC is divided into slices, and interconnected with each core by a ring bus.

The first level of cache is indexed by virtual addresses, and the two other levels are indexed by physical addresses. With caches that implement a *direct addressing* scheme, memory addresses can be decomposed in three parts: the tag, the set and the offset in the line. The lowest $\log_2$(line size) bits determine the offset in the line. The next $\log_2$(number of sets) bits determine the set. The remaining bits form the tag.

The LLC is divided into as many slices as cores, interconnected by a ring bus. The slices contain sets like the other levels. An undocumented hashing algorithm determines the slice associated to an address in order to distribute traffic evenly among the slices and reduce congestion. In contrast to direct addressing, it is a *complex addressing* scheme. Potentially all address bits are used to determine the slice, excluding the lowest $\log_2$(line size) bits that determine the offset in a line. Contrary to the slices, the sets are directly addressed. Figure 2 gives a schematic description of the addressing of slices and sets.

2.2 Cache Attacks

System memory protection prevents a process from directly reading or writing in the cache memory of another process. However, cache hits are faster than cache misses. Thus by monitoring its own activity, *i.e.*, the variation of its own cache access delays, a process can determine the cache sets accessed by other processes, and subsequently leak information. This class of cache attacks is called *access-driven attacks.*

In a *prime+probe* attack [26, 25, 30, 23], the attacker fills the cache, then waits for the victim to evict some cache sets. The attacker reads data again and determines which sets were evicted. The access to these sets will be slower for the attacker because they need to be reloaded in the cache.

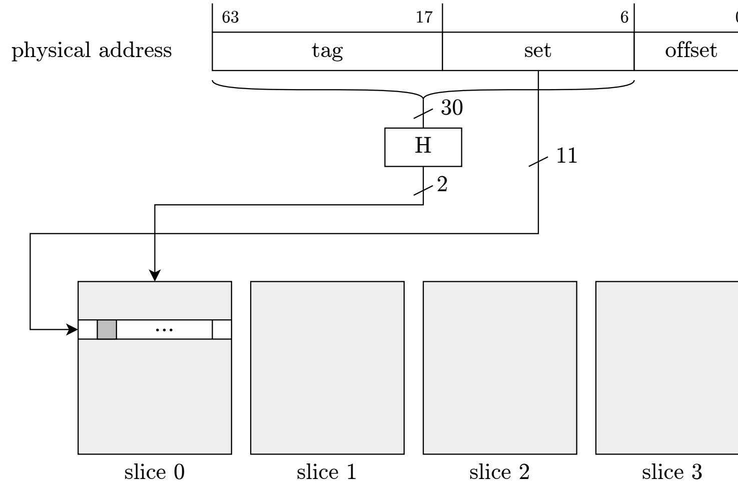

Fig. 2: Complex addressing scheme in the LLC with 64B cache lines, 4 slices and 2048 sets per slice. The slice is given by a hash function that takes as an input all the bits of the set and the tag. The set is then directly addressed. The dark gray cell corresponds to one cache line.

The challenge for this type of fine-grained attack is the ability to target a specific set. This is especially difficult when the targeted cache levels are physically indexed and use complex addressing.

2.3 Hardware Performance Counters

Hardware performance counters are special-purpose registers that are used to monitor special hardware-related events. Such events include cache misses or branch mispredictions, making the counters useful for performance analysis or fine tuning. Because performance counters require high level of privileges, they cannot be directly used for an attack.

The registers are organized by performance monitoring units (called PMON). Each PMON unit has a set of *counter registers*, paired with *control registers*. Performance counters can only be used to measure the global events that happen at the hardware level, and not for a process in particular. This adds noise and has to be considered when performing a measurement.

There is one PMON unit, called CBo (or C-Box), per LLC slice. Each CBo has a separate set of counters, paired to control registers. Among the available events, `LLC_LOOKUP` counts all accesses to the LLC. A mask on the event filters the type of the request (data read, write, external snoop, or any) [9, 11, 12].

Performance counters depend on the processor, but the CBo counters and the `LLC_LOOKUP` event are present in a wide range of processors, and documented

Table 1: Mapping table obtained after running Algorithm 1. Each address has been polled 10000 times.

Physical address	**CBo 0**	**CBo 1**	**CBo 2**	**CBo 3**	**Slice**
0x3a0071010	**11620**	1468	1458	143	0
0x3a0071050	626	**10702**	696	678	1
0x3a0071090	498	567	**10559**	571	2
0x3a00710d0	517	565	573	**10590**	3
...	...	...	...	...	...

by Intel.[3] Some adaptations are needed between different types of processors. Indeed, for Xeon Sandy Bridge, Xeon Ivy Bridge, Xeon Haswell and Core processors, the MSR addresses and the bit fields (thus the values assigned to each MSR) vary, but the method remains similar. Reading and writing MSR registers needs to be done by the kernel (ring 0) via the privileged instructions `rdmsr` and `wrmsr`.

3 Mapping Physical Addresses to Slices Using Performance Counters

In this section, we present our technique for reverse engineering the complex addressing function, using the performance counters. Our objective is to build a table that maps a physical address (for each cache line) to a slice (e.g., Table 1).

This is performed using Algorithm 1. First, monitoring of the `LLC_LOOKUP` event is set up by writing to control registers (MSR). Then, one memory address is repeatedly accessed (Listing 1.1) to generate activity on the corresponding slice. The counter performance registers are then read for each slice (each CBo). Next, the virtual address is translated to a physical address by reading the file `/proc/pid/pagemap`. Finally, the physical address is associated to the slice that has the most lookups. Such monitoring sessions are iterated with different addresses to obtain a set of pairs (physical address, slice) that, eventually, forms a table.

The number of times the address needs to be polled is determined experimentally to differentiate the lookup of this particular address in a slice from the noise of other LLC accesses. We empirically found that polling an address 10 000 times is enough to distinguish the correct slice from noise without ambiguity, and to reproduce the experiment on different configurations. The polling itself is carefully designed to avoid access to memory locations other than the tested address (see Listing 1.1). To this end, most of the variables are put in registers, and the only access to main memory is performed by the `clflush` instruction that flushes the line (in all cache hierarchies). The `clflush` instruction causes a lookup in the LLC even when the line is not present.

[3] For the Xeon range (servers): processors of the micro-architecture Sandy Bridge in [9], Ivy Bridge in [11], and Haswell in [12]. For the Core range (mobiles and workstations), in [10] for the three aforementioned micro-architectures.

Algorithm 1 Constructing the address to slice mapping table.

```
1:  mapping ← new table
2:  for each addr do
3:      for each slice do
4:          write MSRs to set up monitoring LLC_LOOKUP event
5:      end for
6:      polling(addr)                                  // see Listing 1.1
7:      for each slice do
8:          read MSRs to access LLC_LOOKUP event counter
9:      end for
10:     paddr ← translate_address(addr)
11:     find slice i that has the most lookups
12:     insert (paddr, i) in mapping
13: end for
```

Listing 1.1 Memory polling function.

```
1: void polling(uintptr_t addr){
2:   register int i asm ("eax");
3:   register uintptr_t ptr asm ("ebx") = addr;
4:   for(i=0; i<NB_LOOP; i++){
5:     clflush((void*)ptr);
6:   }
7: }
```

Table 2 shows the characteristics of the CPUs we tested. Scanning an address per cache line, *i.e.*, an address every 64B, takes time, but it is linear with the memory size. Scanning 1GB of memory takes a bit less than 45 minutes. We now estimate the storage cost of the mapping table. The lowest 6 bits of the address are used to compute the offset in a line, hence we do not need to store them. In practice, it is also not possible to address all the higher bits because we are limited by the memory available in the machine. For a processor with c slices, the slice is represented with $\lceil log_2(c) \rceil$ bits. A configuration of e.g., 256GB ($= 2^{38}$) of memory and 8 cores can be represented as a table with an index of 32 ($= 38 - 6$) bits; each entry of the table contains 3 bits identifying the slice and an additional bit indicating whether the address has been probed or not. The size of the table is thus $2^{32} \times 4$ bits = 2GB.

Note that the attacker does not necessarily need the entire table to perform an attack. Only the subset of addresses used in an attack is relevant. This subset can be predefined by the attacker, e.g., by fixing the bits determining the set. Alternatively, the subset can be determined dynamically during the attack, and the attacker can query an external server to get the corresponding slice numbers.

Table 2: Characteristics of the Intel CPUs used in our experimentations (mobile and server range).

Name	Model	μ-arch	Cores	Mem
config_1	Xeon E5-2609 v2	Ivy Bridge	4	16GB
config_2	Xeon E5-2660	Sandy Bridge	8	64GB
config_3	Xeon E5-2650	Sandy Bridge	8	256GB
config_4	Xeon E5-2630 v3	Haswell	8	128GB
config_5	Core i3-2350M	Sandy Bridge	2	4GB
config_6	Core i5-2520M	Sandy Bridge	2	4GB
config_7	Core i5-3340M	Ivy Bridge	2	8GB
config_8	Core i7-4810MQ	Haswell	4	8GB
config_9	Xeon E5-2640	Sandy Bridge	6	64GB

4 Building a Compact Addressing Function

4.1 Problem Statement

We aim at finding a function, as a compact form of the table. The function takes n bits of a physical address as input parameters. In the remainder, we note b_i the bit i of the address. The function has an output of $\lceil log_2(c) \rceil$ bits for c slices. To simplify the expression and the reasoning, we express the function as several Boolean functions, one per bit of output. We note $o_i(b_{63}, \ldots, b_0)$ the function that determines the bit i of the output.

Our problem is an instance of Boolean function minimization: our mapping can be seen as a truth table, that can consequently be converted to a formula in Disjunctive Normal Form (DNF). However, the minimization problem is known as NP-hard, and is thus computationally difficult [4].

Existing work on Boolean function minimization does not seem suitable to reconstruct the function from this table. Exact minimization algorithms like Karnaugh mapping or Quine-McCluskey have an exponential complexity in number of input bits. In practice those are limited to 8 bits of input, which is not enough to compute a complete function. The standard tool for dealing with a larger number of inputs is Espresso, which relies on non-optimal heuristics. However, it does not seem suited to handle truth tables of hundreds of millions of lines in a reasonable time.[4] It also gives results in DNF, which won't express the function compactly if it contains logical gates other than AND or OR. Indeed, we provided lines for a subset of the address space to Espresso, but the functions obtained were complex and we did not succeed to generalize them manually. They were generated from a subset, thus they are only true for that subset and do not apply to the whole address space.

We thus need hints on the expression of the function to build a compact addressing function. We did this by a first manual reconstruction, then followed

[4] At the time of camera ready, Espresso has been running without providing any results for more than 2000 hours on a table of more than 100.000.000 lines, which only represents the sixth of the 64GB of memory of the machine.

by a generalization. We have done this work for processors with 2^n cores, which we consider in the remainder of the section.

4.2 Manually Reconstructing the Function for Xeon E5-2609 v2

We now explain how one can manually reverse engineer a complex addressing function: this is indeed how we started for a Xeon E5-2609 v2 (*config_1* in Table 2). In Section 4.3, we will explain how this can be automated and generalized to any processor model with 2^n cores. The following generalization removes the need to perform the manual reconstruction for each setup.

We manually examined the table to search patterns and see if we can deduce relations between the bits and the slices. We performed regular accesses to addresses which were calculated to fix every bit but the ones we want to observe, e.g., regular accesses every 2^6 bytes to observe address bits $b_{11} \dots b_6$. For bits $b_{11} \dots b_6$, we can observe addresses in 4kB pages. For the higher bits (b_{12} and above) we need contiguous physical addresses in a bigger range to fix more bits. This can be done using a custom driver [8], but for implementation convenience we used 1GB pages. Across the table, we observed patterns in the slice number, such as the sequences (0,1,2,3), (1,0,3,2), (2,3,0,1), and (3,2,1,0). These patterns are associated with the XOR operation of the input bits, this made the manual reconstruction of the function easier.

We obtained these two binary functions:

$$\begin{aligned} o_0(b_{63}, \dots, b_0) = b_6 \oplus b_{10} \oplus b_{12} \oplus b_{14} \oplus b_{16} \oplus b_{17} \oplus b_{18} \oplus b_{20} \oplus b_{22} \oplus b_{24} \oplus b_{25} \\ \oplus b_{26} \oplus b_{27} \oplus b_{28} \oplus b_{30} \oplus b_{32} \oplus b_{33}. \end{aligned}$$

$$\begin{aligned} o_1(b_{63}, \dots, b_0) = b_7 \oplus b_{11} \oplus b_{13} \oplus b_{15} \oplus b_{17} \oplus b_{19} \oplus b_{20} \oplus b_{21} \oplus b_{22} \oplus b_{23} \oplus b_{24} \\ \oplus b_{26} \oplus b_{28} \oplus b_{29} \oplus b_{31} \oplus b_{33} \oplus b_{34}. \end{aligned}$$

We confirmed the correctness of the obtained functions by comparing the output of the slice calculated with the function against the entire mapping table obtained with the MSRs.

4.3 Reconstructing the Function Automatically

Our manual reconstruction shows that each output bit $o_i(b_{63}, \dots, b_0)$ can be expressed as a series of XORs of the bits of the physical address. Hund et al. [8] manually reconstructed a mapping function of the same form, albeit a different one. In the remainder, we thus hypothesize, and subsequently validate the hypothesis, that the function has the same form for all processors that have 2^n cores.

The fact that the function only relies on XORs makes its reconstruction a very constrained problem. For each Boolean function $o_i(b_{63}, \dots, b_0)$, we can analyze the implication of the address bits independently from each other, in order

Table 3: Functions obtained for the Xeon and Core processors with 2, 4 and 8 cores. Gray cells indicate that a machine with more memory would be needed to determine the remaining bits.

		Address Bit																															
		37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06
2 cores	o_0						$\oplus$		$\oplus$		$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$		$\oplus$		$\oplus$		$\oplus$	$\oplus$	$\oplus$		$\oplus$		$\oplus$		$\oplus$				$\oplus$
4 cores	o_0					$\oplus$	$\oplus$		$\oplus$		$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$		$\oplus$		$\oplus$		$\oplus$	$\oplus$	$\oplus$		$\oplus$		$\oplus$		$\oplus$				$\oplus$
	o_1				$\oplus$	$\oplus$		$\oplus$		$\oplus$	$\oplus$		$\oplus$		$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$		$\oplus$		$\oplus$		$\oplus$		$\oplus$				$\oplus$	
8 cores	o_0		$\oplus$	$\oplus$		$\oplus$	$\oplus$		$\oplus$		$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$		$\oplus$		$\oplus$		$\oplus$	$\oplus$	$\oplus$		$\oplus$		$\oplus$		$\oplus$				$\oplus$
	o_1	$\oplus$		$\oplus$	$\oplus$	$\oplus$		$\oplus$		$\oplus$	$\oplus$		$\oplus$		$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$		$\oplus$		$\oplus$		$\oplus$		$\oplus$				$\oplus$	
	o_2	$\oplus$	$\oplus$	$\oplus$	$\oplus$			$\oplus$	$\oplus$			$\oplus$	$\oplus$			$\oplus$	$\oplus$			$\oplus$			$\oplus$			$\oplus$	$\oplus$				$\oplus$		

to access only a handful of physical addresses. Our algorithm finds two addresses that only differ by one bit, finds their respective slices using performance counters, and compares the output. If the output is the same, it means that the bit is not part of the function. Conversely, if the output differs, it means that the bit is part of the function. Note that this only works for a XOR function. This algorithm is linear in number of bits.

To implement the algorithm, we use huge pages of 1GB on Xeon processors (resp. 2MB on Core processors), which is contiguous physical memory naturally aligned on the huge page size. The offset in a huge page is 30-bit (resp. 21-bit) long, therefore the lowest 30 bits (resp. 21 bits) in virtual memory will be the same as in physical memory. We thus calculate offsets in the page that will result in physical addresses differing by a bit, without converting virtual addresses to physical addresses. To discover the remaining bits, we allocate several huge pages, and convert their base virtual address to physical address to find those that differ by one bit. In order to do this, we allocate as many huge pages as possible.

To evaluate the algorithm, we retrieved the function for all models from *config_1* to *config_8* of Table 2, results are summarized in Table 3. The functions are given for the machine that has the most memory, to cover as many bits as possible. We remark that the functions, for a given number of cores, are identical among all processors, for all ranges of products and micro-architectures. Using the above mentioned algorithm, we obtained those functions quickly (from a few seconds to five minutes in the worst cases). We also remark that we have in total 3 functions o_0, o_1 and o_2 for all processors, and that the functions used only depends on the number of cores, regardless of the micro-architecture or the product range. While in retrospective this seems to be the most straightforward solution to be adopted by Intel, this was far from evident at the beginning of our investigations. Now that the functions are known, an attacker can use them to perform his attacks without any reverse engineering.

5 Using the Function to Build a Covert Channel

To verify empirically the correctness of the function, we build a covert channel. This covert channel uses similar principles to the one of Maurice et al. [22]. It is

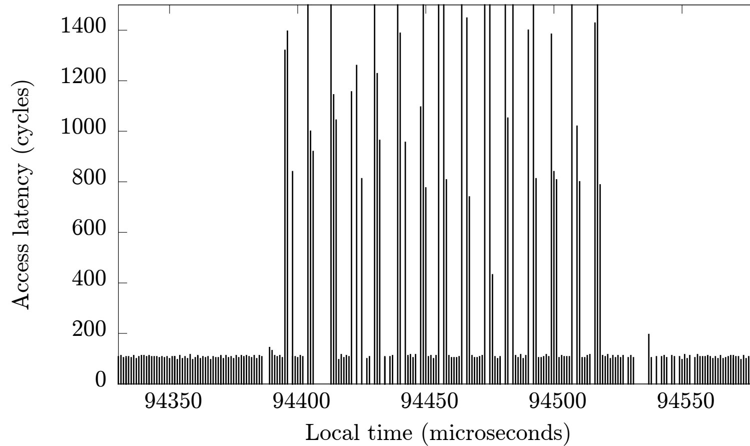

Fig. 3: Receiving interleaved ‘0’s and ‘1’s to test the raw bitrate of the covert channel.

based on the fact that the LLC is inclusive, *i.e.*, when a line is evicted from the LLC, it is also evicted from the L1 and L2. With this property, a program on any core can evict a line from the private cache of another core. This property can then be used by two programs to communicate. The work in [22] bypasses the complex addressing issue by evicting the whole LLC. However, the LLC typically stores a few megabytes, and thus the sender needs to access a buffer that is the size of (or bigger than) the LLC to evict it entirely. Having the complex addressing function, the sender targets a set in a slice, and thus evicts a cache line with much fewer accesses. For example, in the case of a 12-way associative LLC, assuming a pseudo-LRU replacement policy, the sender needs approximately 12 accesses to evict the whole set.

In this covert channel, the sender creates a set of physical addresses that map to the same set, using the function and the translation from virtual to physical addresses. It repeatedly accesses these addresses to send a ‘1’, and does nothing to send a ‘0’. The receiver has a set of physical addresses that map to the same LLC set as the sender’s. When the sender sends a ‘1’, it evicts the data of the receiver from the LLC, and thus from its private L1 cache. The receiver consequently observes a slow access to its set.

We conduct an experiment on *config_1* to estimate the bitrate of this covert channel, in which the sender transmits interleaved ‘0’s and ‘1’s. Figure 3 illustrates the measurements performed by the receiver. According to the measurements, 29 bits can be transmitted over a period of 130 microseconds, leading to a bitrate of approximately 223 kilobits per second.

6 Discussion

6.1 Dealing with Unknown Physical Addresses

The translation from virtual to physical addresses is unknown to the attacker in most practical setups, like in virtualized or sandboxed environments. We now describe a possible extension to the covert channel described in Section 5 to avoid using this address translation.

Similarly to the work of Liu et al. [20] and Irazoqui et al. [14], the sender and the receiver both use huge pages. The cache index bits are thus the same for virtual and physical addresses. Using the function only on the bits in the offset of the huge page, the sender is able to create a set of addresses that map to the same set, in the same slice. As some bits of the physical address are unknown, he does not know the precise slice. However, he does know that these addresses are part of a single set, in a single slice.

The receiver now performs the same operation. The receiver has only the knowledge of the index set to target, but he does not know in which of the n slices. He thus creates n sets of addresses, each one being in a different slice. He then continuously accesses each of these sets. The receiver will only receive transmitted bits in a single set: from now on, he can target a single set. The sender and the receiver are effectively accessing the same LLC set in the same slice.

6.2 Other Applications

Reverse engineering the complex addressing function is orthogonal to performing cache attacks. Indeed, knowing the correct addressing function can help any fine-grained attack on the LLC. Cache attacks rely on the attacker evicting data from a cache level. This can be done by the `clflush` instruction. However, it requires shared memory in a covert or side channel scenario, and it is not available in all environments. We thus focus on building attacks without this instruction. To perform an attack on the LLC, the attacker needs to create an eviction set, and to subsequently access the data to evict the lines that are currently cached. There are two methods to find an eviction set: a *dynamic* approach based on a timing attack that does not require the function, and a *static* approach that uses the function to compute addresses that belong to an eviction set. Building a static eviction set has the advantage of being faster than building a dynamic one. Indeed, the function is already known, whereas the dynamic set has to be computed for each execution. Moreover, Gruss et al. [6] showed that dynamically computing a set to achieve an optimal eviction is a slow operation.

Hund et al. [8] defeated KASLR using the static approach. Similarly, Irazoqui et al. [14] used a static approach on a Nehalem CPU that does not use complex addressing. Yet, their attack requires understanding the slice selection, and thus having the complex addressing function for more recent CPUs. Concurrently to this work, Gruss et al. [6] used the complex addressing function to build a proof-of-concept of the *rowhammer* attack without the `clflush` instruction.

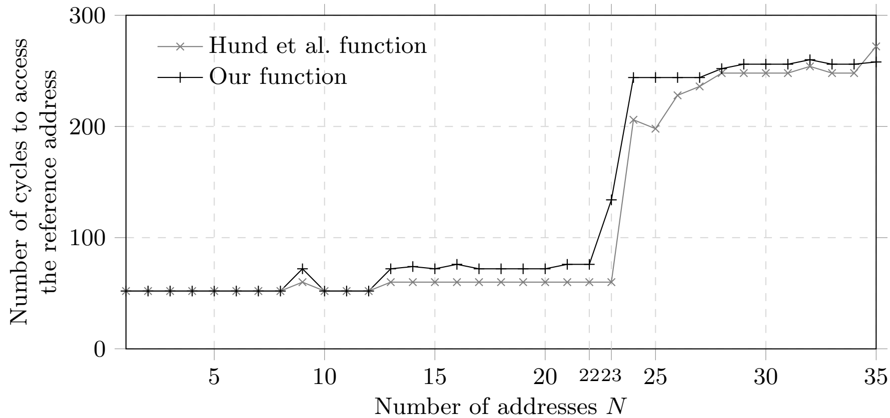

Fig. 4: Median number of cycles to access a reference address, after accessing N addresses in the same set, which is calculated using [8] and our function. Results on 100 runs, on $config_1$ (Ivy Bridge with a 20-way associative LLC).

Rowhammer is not a typical cache attack, since it exploits a bug on the DRAM to flip bits. However, the bug is triggered by frequent accesses to the DRAM, *i.e.*, non-cached accesses. The original exploits used the `clflush` instruction, that is not available in e.g., Javascript. An attack that seeks to avoid using the `clflush` instruction thus also needs to compute an eviction set.

6.3 Comparison to Previously Retrieved Functions

We observe that the functions we obtained differ from the ones obtained by Hund et al. [8], and Seaborn [29]. In particular, Hund et al. found that the functions only use the tag bits (bits b_{17} to b_{31}). We argue that their method does not infer the presence of the bits used to compute the set (bits b_6 to b_{16}). Indeed, as they searched for colliding addresses, they obtained addresses that belong to the same slice and the same set. As in this case the set is directly mapped to the bits b_6 to b_{16}, addresses that collide have the same values for these bits. Therefore, if the function that computes the slice uses the bits b_6 to b_{16}, the method of [8] is not able to retrieve them. On the contrary, our method retrieves the slices regardless of the sets, leading to a complete function.

We also observe that the function we retrieved for 2 cores is the same as the one retrieved in [29], albeit a more complete one. However, the function we retrieve for 4 cores does not use the same bits as the one retrieved in [8]. We argue that we do have access to the ground truth (*i.e.*, the slices accessed), whereas they rely on *indirect measurements*. Several correct functions can however be retrieved, as the slices can be labeled differently from one work to another.

To compare our function against [8], we performed the following experiment. Using the retrieved addressing function, we constructed a set of physical addresses that are supposed to map the same set (thus the same slice). We accessed

N different addresses from this set. We then measured the access time to the first reference address accessed, to see if it was evicted from the cache. Figure 4 shows the median number of CPU cycles to access the reference address for different values of N, for 100 runs. The function that is the most precise should have a memory access time spike the closest $N = 20$ (which is the cache associativity). We observe that both functions have a spike slightly after $N = 20$. We note that the spike occurs for a value $N > 20$ and not exactly $N = 20$: it is most likely due to the fact that the replacement policy on Ivy Bridge is not strictly LRU, but a variant called Quad-Age LRU [18]. In practice, both functions are able to evict a cache line with few accesses. However, our function seems more precise than the one of [8], leading to fewer accesses to evict a cache line ($N = 23$ accesses for our function, $N = 24$ for [8]), and a sharper transition. This also confirms the correctness of our function.

7 Related work

Hardware performance counters are traditionally used for performance monitoring. They have also been used in a few security scenarios. In defensive cases, they are used to detect an anomalous behavior such as malware detection [5], integrity checking of programs [21], control flow integrity [34], and binary analysis [32]. Uhsadel et al. [31] used performance counters in offensive cases to profile the cache and derive a side-channel attack against AES. However, the performance counters can only be read with high privileges, *i.e.*, in kernel-mode, or being root in user-mode if a driver is already loaded. Contrary to this attack, we use performance counters to infer hardware properties offline, and our subsequent cache attack does not need high privileges.

The Flush+Reload attack [36] relies on shared memory, and more precisely on shared libraries, to evict lines of cache, using the `clflush` instruction. In this attack, the attacker leverages the shared and inclusive LLC to run concurrently to the victim on separate cores, including on separate virtual machines. Flush+Reload has been used to attack implementations of RSA [36], AES [16] and ECDSA [1]. It has also been used to find a new side channel that revives a supposedly fixed attack on CBC encryption mode [13], and to detect cryptographic libraries [17]. Gruss et al. [7] presented a generic technique to profile and exploit cache-based vulnerabilities, using Flush+Reload. Memory sharing can be easily disabled in virtualized environments (which is already the case in the cloud environment), effectively rendering impossible the Flush+Reload attack. On sandboxed environments, like Javascript or Native Client [2], the ability to perform the Flush+Reload attack is also compromised by the absence of `clflush` instruction [3]. Understanding how complex addressing works allows performing cache attacks in these environments, without the need of shared memory or `clflush`.

Simultaneously to our work, Irazoqui et al. [14], Liu et al. [20], and Oren et al. [24] have extended the Prime+Probe attack to the LLC. They are thus able to perform side channels on the LLC without any shared memory. They construct a

set of addresses that map to the same set as the line to evict. Irazoqui et al. [14] have used a Nehalem processor that does not use complex addressing. Therefore huge pages are sufficient to construct this set of addresses. Liu et al. [20], and Oren et al. [24] targeted more recent processors that use complex addressing. They, however, performed their attacks without reverse engineering the complex addressing function. Thus, even if we share the same motivation, *i.e.*, performing cache attacks on recent processors without any shared memory, our works are very different in their approaches. We also note that our work has a broader application, as it contributes to a better understanding of the undocumented complex addressing function, possibly leading to other types of attacks.

Other work is directly interested in retrieving the function, and several attempts have been made to reverse engineer it. Hund et al. [8] performed a manual reverse engineering for a 4-core Sandy Bridge CPU. Their method uses colliding addresses, *i.e.*, *indirect measurements*, to derive the function. They used this function to bypass kernel ASLR. Very recently, and also simultaneous to our work, Seaborn [29] continued the work of [8], with a 2-core Sandy Bridge CPU. The intended goal is to exploit the *rowhammer* bug with cached accesses, without the `clflush` instruction. Gruss et al. [6] subsequently demonstrated a *rowhammer* attack on Javascript, using the complex addressing function. In contrast, we do not use the same method to retrieve the addressing function as [8, 29]. Our method, using performance counters, performs *direct measurements*, *i.e.*, retrieves the exact slice for each access. We thus show that the functions in [8, 29] are partially incorrect, even though they are sufficient to be used in practice. We also derive a function for all processors with 2^n cores, automating the reverse engineering. Different from these two works, we also have tested our method on a large variety of processors. Concurrently to our work, Irazoqui et al. [15] worked on automating this reverse engineering, and evaluated their work on several processors. However, their method is similar to Hund et al. [8], and thus suffers from the same limitations.

8 Conclusions

In this paper, we introduced a novel method to reverse engineer Intel's undocumented complex addressing, using hardware performance counters. The reversed functions can be exploited by an attacker to target specific sets in the last level cache when performing cache attacks. Contrary to previous work, our method is automatic, and we have evaluated it on a wide range of processors, for different micro-architectures, numbers of cores, and product ranges. We also obtained a more complete and more correct description of the complex addressing function than previous work, *i.e.*, taking into account more bits of the memory address. In the general case with any number of cores, we automatically built a table that maps physical addresses to cache slices. This table already enables to perform targeted cache attacks but may require an important amount of storage. In the case of CPUs with 2^n cores we provided a compact function that maps addresses to slices, rendering the attacks even more effective. We demonstrated

a covert channel to prove the correctness of our function, and discussed other applications such as exploiting the *rowhammer* bug in Javascript.

Our work expands the understanding of these complex and only partially documented pieces of hardware that are modern processors. We foresee several directions for future work. First, a compact representation for CPUs with a number of cores different from 2^n would generalize our findings. Second, we believe that new attacks could be made possible by knowing the complex addressing of a cache. Finally, we believe that understanding the complex addressing function enables the development of countermeasures to cache attacks.

Acknowledgments

We would like to thank Mark Seaborn, Mate Soos, Gorka Irazoqui, Thomas Eisenbarth and our anonymous reviewers for their valuable comments and suggestions. We also greatly thank Stefan Mangard and Daniel Gruss for the collaboration on the exploitation of the *rowhammer* bug in Javascript, for which we applied the findings of this article after its submission.

References

1. N. Benger, J. van de Pool, N. P. Smart, and Y. Yarom. "Ooh Aah... Just a Little Bit" : A small amount of side channel can go a long way. In *Proceedings of the 16th Workshop on Cryptographic Hardware and Embedded Systems (CHES'14)*, pages 75–92, 2014.
2. Chrome Developers. Native Client. `https://developer.chrome.com/native-client`. Retrieved on June 2, 2015.
3. Chrome Developers. Native Client Revision 13809. `http://src.chromium.org/viewvc/native_client?revision=13809&view=revision`, September 2014. Retrieved on June 2, 2015.
4. Y. Crama and P. L. Hammer. *Boolean functions: Theory, algorithms, and applications*. Cambridge University Press, 2011.
5. J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo. On the feasibility of online malware detection with performance counters. *ACM SIGARCH Computer Architecture News*, 41(3):559–570, 2013.
6. D. Gruss, C. Maurice, and S. Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. *arXiv:1507.06955v1*, 2015.
7. D. Gruss, R. Spreitzer, and S. Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *Proceedings of the 24th USENIX Security Symposium*, 2015.
8. R. Hund, C. Willems, and T. Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (S&P'13)*, pages 191–205. Ieee, May 2013.
9. Intel. Intel ® Xeon ® Processor E5-2600 Product Family Uncore Performance Monitoring Guide. 327043-001:1–136, 2012.
10. Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide. 3(253665), 2014.

11. Intel. Intel® Xeon® Processor E5 v2 and E7 v2 Product Families Uncore Performance Monitoring Reference Manual. 329468-002:1—-200, 2014.
12. Intel. Intel® Xeon® Processor E5 v3 Family Uncore Performance Monitoring Reference Manual. 331051-001:1–232, 2014.
13. G. Irazoqui, T. Eisenbarth, and B. Sunar. Lucky 13 Strikes Back. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (AsiaCCS'15)*, pages 85–96, 2015.
14. G. Irazoqui, T. Eisenbarth, and B. Sunar. S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing—and its Application to AES. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)*, 2015.
15. G. Irazoqui, T. Eisenbarth, and B. Sunar. Systematic Reverse Engineering of Cache Slice Selection in Intel Processors. In *Proceedings of the 18th EUROMICRO Conference on Digital System Design*, 2015.
16. G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar. Wait a minute! A fast, Cross-VM attack on AES. In *Proceedings of the 17th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'14)*, 2014.
17. G. Irazoqui, M. S. IncI, T. Eisenbarth, and B. Sunar. Know Thy Neighbor: Crypto Library Detection in Cloud. *Proceedings on Privacy Enhancing Technologies*, 1(1):25–40, 2015.
18. S. Jahagirdar, V. George, I. Sodhi, and R. Wells. Power Management of the Third Generation Intel Core Micro Architecture formerly codenamed Ivy Bridge. Hot Chips 2012, `http://hotchips.org/wp-content/uploads/hc_archives/hc24/HC24-1-Microprocessor/HC24.28.117-HotChips_IvyBridge_Power_04.pdf`. Retrieved on July 16, 2015.
19. D.-h. Kim, P. J. Nair, and M. K. Qureshi. Architectural Support for Mitigating Row Hammering in DRAM Memories. *IEEE Computer Architecture Letters*, 14(1):9–12, 2014.
20. F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)*, 2015.
21. C. Malone, M. Zahran, and R. Karri. Are Hardware Performance Counters a Cost Effective Way for Integrity Checking of Programs. In *Proceedings of the sixth ACM workshop on Scalable Trusted computing*, 2011.
22. C. Maurice, C. Neumann, O. Heen, and A. Francillon. C5: Cross-Cores Cache Covert Channel. In *Proceedings of the 12th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, July 2015.
23. M. Neve and J.-P. Seifert. Advances on Access-Driven Cache Attacks on AES. In *Proceedings of the 13th international conference on Selected areas in cryptography (SAC'06)*, pages 147–162, 2006.
24. Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS'15)*, 2015.
25. D. A. Osvik, A. Shamir, and E. Tromer. Cache Attacks and Countermeasures: the Case of AES. In *Proceedings of the Cryptographers' Track at the RSA Conference (CT-RSA 2006)*, pages 1–25, 2006.
26. C. Percival. Cache missing for fun and profit. In *Proceedings of BSDCan*, pages 1–13, 2005.
27. T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *Pro-*

ceedings of the 16th ACM conference on Computer and Communications Security (CCS'09), pages 199–212, 2009.
28. M. Seaborn. Exploiting the DRAM rowhammer bug to gain kernel privileges. `http://googleprojectzero.blogspot.fr/2015/03/exploiting-dram-rowhammer-bug-to-gain.html`, March 2015. Retrieved on June 2, 2015.
29. M. Seaborn. L3 cache mapping on Sandy Bridge CPUs. `http://lackingrhoticity.blogspot.fr/2015/04/l3-cache-mapping-on-sandy-bridge-cpus.html`, April 2015. Retrieved on June 2, 2015.
30. E. Tromer, D. A. Osvik, and A. Shamir. Efficient Cache Attacks on AES, and Countermeasures. *Journal of Cryptology*, 23(1):37–71, July 2010.
31. L. Uhsadel, A. Georges, and I. Verbauwhede. Exploiting hardware performance counters. In *Proceedings of the 5th International Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2008)*, pages 59–67, 2008.
32. C. Willems, R. Hund, A. Fobian, D. Felsch, and T. Holz. Down to the bare metal: Using processor features for binary analysis. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC'12)*, pages 189–198, 2012.
33. Z. Wu, Z. Xu, and H. Wang. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In *Proceedings of the 21st USENIX Security Symposium*, 2012.
34. Y. Xia, Y. Liu, H. Chen, and B. Zang. CFIMon: Detecting violation of control flow integrity using performance counters. In *Proceedings of the 42th International Conference on Dependable Systems and Networks (DSN'12)*, pages 1–12, 2012.
35. Y. Xu, M. Bailey, F. Jahanian, K. Joshi, M. Hiltunen, and R. Schlichting. An exploration of L2 cache covert channels in virtualized environments. In *Proceedings of the 3rd ACM Cloud Computing Security Workshop (CCSW'11)*, pages 29–40, 2011.
36. Y. Yarom and K. Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proceedings of the 23th USENIX Security Symposium*, 2014.
37. Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 19th ACM conference on Computer and Communications Security (CCS'12)*, 2012.
38. Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS'14)*, pages 990–1003, New York, New York, USA, 2014. ACM Press.